Common Lisp Quick Reference

(asinh a) ➤ asinh a, acosh a, or atanh a, respectively.
(acosh a) ➤ Return $e^{a^2}$ = cos a + i sin a.
(atanh a) ➤ Return complex conjugate of a.

(f cis a) ➤ Return $e^{ia} = \cos a + i \sin a$.
(f conjugate a) ➤ Return complex conjugate of a.

(f max num+) ➤ Greatest or least, respectively, of nums.
(f min num+) ➤ Greatest or least, respectively, of nums.

\[
\begin{cases}
\text{round} & n \lfloor n \rfloor \\
floor & n \lfloor n \rfloor \\
\text{ceiling} & n \lceil n \rceil \\
\text{truncate} & n \lfloor n \rfloor
\end{cases}
\]

(n \lfloor n \rfloor) ➤ Return as integer or float, respectively, n/d rounded, or rounded towards $-\infty$, $+\infty$, or 0, respectively; and remainder.

(f mod n d) ➤ Same as f floor or f truncate, respectively, but return remainder only.

(random limit [state [random-state]]) ➤ Return non-negative random number less than limit, and of the same type.

(make-random-state [state [NIL T]]) ➤ Copy of random-state object state or of the current random state; or a randomly initialized fresh random state.

(random-state*) ➤ Current random state.

(float-sign num-a [num-b]) ➤ num-b with num-a's sign.

(signum n) ➤ Number of magnitude 1 representing sign or phase of n.

(numerator rational) ➤ Numerator or denominator, respectively, of rational's canonical form.

(denominator rational) ➤ Numerator or denominator, respectively, of rational's canonical form.

(realpart number) ➤ Real part or imaginary part, respectively, of number.

(imagpart number) ➤ Make a complex number.

(phase num) ➤ Angle of num's polar representation.

(abs n) ➤ Return |n|.

(rational real) ➤ Convert real to rational. Assume complete/limited accuracy for real.

(rationalize real) ➤ Convert real into float with type of prototype.

1.3 Logic Functions

Negative integers are used in two's complement representation.

(f boole operation int-a int-b) ➤ Return value of bitwise logical operation. operations are

\[
\begin{align*}
\text{boole-1} & : \text{int-a} \\
\text{boole-2} & : \text{int-b} \\
\text{boole-c1} & : \neg \text{int-a} \\
\text{boole-c2} & : \neg \text{int-b} \\
\text{boole-set} & : \text{All bits set} \\
\text{boole-clr} & : \text{All bits zero}
\end{align*}
\]
1 Numbers

1.1 Predicates

(i= number+)  \( \Rightarrow T \) if all numbers, or none, respectively, are equal in value.

(i /= number+) \( \Rightarrow T \) if all numbers are not equal, respectively.

(i > number+) \( \Rightarrow T \) if all numbers are greater than the specified number, respectively.

(i <= number+) \( \Rightarrow T \) if all numbers are greater than or equal to the specified number, respectively.

(i < number+) \( \Rightarrow T \) if all numbers are less than the specified number, respectively.

(i <= number+) \( \Rightarrow T \) if all numbers are less than or equal to the specified number, respectively.

1.2 Numeric Functions

(i + a b ) \( \Rightarrow \sum a \) or \( b \), respectively.

(i * a b ) \( \Rightarrow \prod a \) or \( b \), respectively.

(i - a b ) \( \Rightarrow a - \sum b \) or \( \prod b \) or \( \prod a \), respectively. Without any bs, return \( -a \) or \( 1/a \), respectively.

(i / a b ) \( \Rightarrow a \) or \( 1/a \), respectively.

(i l + a b ) \( \Rightarrow a + 1 \) or \( a - 1 \), respectively.

(i s n ) \( \Rightarrow \sqrt{n} \) in complex numbers/natural numbers.

(i l c m integer+ ) \( \Rightarrow \text{least common multiple} \) of integers. 

(i g c d integer+ ) \( \Rightarrow \text{greatest common denominator} \), respectively, of integers. 

(i p i ) \( \Rightarrow \text{long-float approximation of} \ \pi \), Ludolph’s number.

(i s in a ) \( \Rightarrow \sin a \), \( \cos a \), or \( \tan a \), respectively. (a in radians.)

(i t a n a ) \( \Rightarrow \text{arcsin} a \) or \( \text{arccos} a \), respectively, in radians.

(i a t a n a ) \( \Rightarrow \arctan \frac{a}{b} \) in radians.

(i s h a ) \( \Rightarrow \sinh a \), \( \cosh a \), or \( \tanh a \), respectively.

(i t a n h a ) \( \Rightarrow \tanh a \), \( \sinh a \), or \( \cosh a \), respectively.
4 Conses

4.1 Predicates

(consp foo)  \(\triangleright\) Return T if foo is of indicated type.

(flistp foo) \(\triangleright\) Return list if list/fo0 is NIL.

(endp list) \(\triangleright\) Return T if list/fo0 is NIL.

(atom foo) \(\triangleright\) Return T if foo is not a cons.

(tailp foo list) \(\triangleright\) Return T if foo is a tail of list.

(member list list) \(\triangleright\) Return tail of list starting with its first element matching foo. Return NIL if there is no such element.

(member-if list list \{key function\}) \(\triangleright\) Return tail of list starting with its first element satisfying test. Return NIL if there is no such element.

(subsetp list-a list-b) \(\triangleright\) Return T if list-a is a subset of list-b.

4.2 Lists

(cons foo bar) \(\triangleright\) Return new cons \{foo, bar\}.

(listp foo) \(\triangleright\) Return list of foos.

(list* foo) \(\triangleright\) Return list of fos with last foo becoming cdr of last cons. Return fo0 if only one foo given.

(make-list num \{initial-element fo0\}) \(\triangleright\) New list with num elements set to fo0.

(length list) \(\triangleright\) Length of list; NIL for circular list.

(car list) \(\triangleright\) Car of list or NIL if list is NIL. setfable.

(cdr list) \(\triangleright\) Cdr of list or NIL if list is NIL. setfable.

(rest list) \(\triangleright\) Cdr of list or NIL if list is NIL. setfable.

(nthcdr n list) \(\triangleright\) Return tail of list after calling cdr n times.

(first \{second \{third \{fourth \{fifth \{sixth \{\ldots \{nth \{tenth\}\}\}\}\}\}\}\}\}\) \(\triangleright\) Return nth element of list if any, or NIL otherwise. setfable.

(nth n list) \(\triangleright\) Zero-indexed nth element of list. setfable.

(cadr list) \(\triangleright\) With X being one to four as and ds representing cars and cdrs, e.g. \{cadr bar\} is equivalent to \{car \{cadr bar\}\}. setfable.

(rlist list \{num\}) \(\triangleright\) Return list of last num conses of list.

1.4 Integer Functions

(integer-length integer) \(\triangleright\) Number of bits necessary to represent integer.

(ldb-test byte-spec integer) \(\triangleright\) Return T if any bit specified by byte-spec in integer is set.

(ash integer count) \(\triangleright\) Return copy of integer arithmetically shifted left by count adding zeros at the right, or, for count < 0, shifted right discarding bits.

(ldb byte-spec integer) \(\triangleright\) Extract byte denoted by byte-spec from integer. setfable.

(deposit-field int-a byte-spec int-b) \(\triangleright\) Return int-b with bits denoted by byte-spec replaced by corresponding bits of int-a, or by the low \{byte-size byte-spec\} bits of int-a, respectively.

(mask-field byte-spec integer) \(\triangleright\) Return copy of integer with all bits set but those denoted by byte-spec. setfable.

(byte size position) \(\triangleright\) Byte specifier for a byte of size bits starting at a weight of position.

(byte-size byte-spec) \(\triangleright\) Size or position, respectively, of byte-spec.
15 Implementation-Dependent

### 1.5 Implementation-Dependent

- **short-float**
- **single-float**
- **double-float**
- **long-float**

  - Smallest possible number making a difference when added or subtracted, respectively.

- **least-negative**
- **least-negative-normalized**
- **least-positive**
- **least-positive-normalized**

  - Available numbers closest to \(-0\) or \(+0\), respectively.

- **most-negative**
- **most-positive**

  - Available numbers closest to \(-\infty\) or \(+\infty\), respectively.

- **(decode-float n)**
- **(integer-decode-float n)**

  - Return significand, exponent, and sign of **float n**.

- **(scale-float n i)**

  - With n’s radix b, return \(nb^i\).

- **(float-radix n)**
- **(float-digits n)**
- **(float-precision n)**

  - Radix, number of digits in that radix, or precision in that radix, respectively, of **float n**.

- **(upgraded-complex-part-type foo [environment]**

  - Type of most specialized **complex** number able to hold parts of type **foo**.

### 2 Characters

The **standard-char** type comprises \(a-z\), \(A-Z\), 0-9, **Newline**, **Space**, and \(!?\#^_*'\;\cdot\;+\-\%\&\()\{}\).

- **(characterp char)**
- **(standard-char-p char)**

  - \(T\) if argument is of indicated type.

- **(graphic-char-p character)**
- **(alpha-char-p character)**
- **(alphabetic-p character)**

  - \(T\) if **character** is visible, alphabetic, or alphanumeric, respectively.

- **(upper-case-p character)**
- **(lower-case-p character)**
- **(both-case-p character)**

  - Return \(T\) if **character** is uppercas, lowercase, or able to be in another case, respectively.

- **(digit-char-p character [radix]**

  - Return its weight if **character** is a digit, or **NIL** otherwise.

- **(char= character**
- **(char/= character**

  - Return \(T\) if all characters, or none, respectively, are equal.

- **(char= character**
- **(char/= character**

  - Return \(T\) if all characters, or none, respectively, are equal ignoring case.

- **(char> character**
- **(char>= character**
- **(char< character**
- **(char<= character**

  - Return \(T\) if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

### 3 Strings

Strings can as well be manipulated by array and sequence functions; see pages 10 and 12.

- **(stringp foo)**
- **(simple-string-p foo)**

  - \(T\) if **foo** is of indicated type.

- **(string= string-string-p foo)**

  - Return \(T\) if subsequences of **foo** and **bar** are equal. **Obey/ignore**, respectively, case.

- **(make-string size [initial-element element-type char]**

  - Return string of length **size**.

- **(string x)**

  - **string-capitalize**
  - **string-upper-case**
  - **string-downcase**

    - Convert \(x\) **symbol**, **string**, or **character** into a string, a string with capitalized words, an all-upper-case string, or an all-lowercase string, respectively.

- **(nstring-capitalize nstring-upper-case nstring-downcase)**

    - Convert string into a string with capitalized words, an all-upper-case string, or an all-lowercase string, respectively.

- **(string-trim**
  - **(string-left-trim**
  - **(string-right-trim**

    - Return string with all characters in sequence **char-bag** removed from both ends, from the beginning, or from the end, respectively.
6 Sequences

6.1 Sequence Predicates

\{(\textit{every} \text{test sequence}^+)\}
\hspace{1em}⇒ \text{Return NIL or T, respectively, as soon as \text{test} on any set of corresponding elements of sequences returns NIL.}

\{(\textit{some} \text{test sequence}^+)\}
\hspace{1em}⇒ \text{Return value of test or NIL, respectively, as soon as \text{test} on any set of corresponding elements of sequences returns non-NIL.}

\{(\textit{notany} \text{test sequence})\}
\hspace{1em}⇒ \text{Return position in \text{sequence-a} where \text{sequence-a} and \text{sequence-b} begin to mismatch. Return NIL if they match entirely.}

\{(\textit{mismatch} \text{sequence-a sequence-b})\}
\hspace{1em}⇒ \text{Return \textit{mismatch} function value.}

6.2 Sequence Functions

\{(\textit{make-sequence} \text{sequence-type} \text{size} [\text{initial-element foo}])\}
\hspace{1em}⇒ \text{Make sequence of sequence-type with size elements.}

\{(\textit{concatenate} \text{type} \text{sequence}^+)\}
\hspace{1em}⇒ \text{Return concatenated sequence of type.}

\{(\textit{merge} \text{type} \text{sequence-a} \text{sequence-b} \text{test} \text{[key function]}\text{test})\}
\hspace{1em}⇒ \text{Return interleaved sequence of type. Merged sequence will be sorted if both sequence-a and sequence-b are sorted.}

\{(\textit{fill} \text{sequence} \text{foo} [\text{start start-end}]\text{[key function]})\}
\hspace{1em}⇒ \text{Return sequence after setting elements between start and end to \text{foo}.}

\{(\textit{length} \text{sequence})\}
\hspace{1em}⇒ \text{Return length of sequence (being value of fill pointer if applicable).}

\{(\textit{count} \text{foo} \text{sequence})\}
\hspace{1em}⇒ \text{Return number of elements in \text{sequence} which match \text{foo}.}

\{(\textit{count-if} \text{test} \text{sequence})\}
\hspace{1em}⇒ \text{Return number of elements in \text{sequence} which satisfy \text{test}.}

\{(\textit{elt} \text{sequence} \text{index})\}
\hspace{1em}⇒ \text{Return element of \text{sequence} pointed to by zero-indexed index. setfable.}

\{(\textit{subseq} \text{sequence} \text{start} \text{end})\}
\hspace{1em}⇒ \text{Return subsequence of \text{sequence} between \text{start} and \text{end}. setfable.}

\{(\textit{sort} \text{sequence} \text{test} \text{[key function]})\}
\hspace{1em}⇒ \text{Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.}

\{(\textit{reverse} \text{sequence})\}
\hspace{1em}⇒ \text{Return \text{sequence} in reverse order.}

\{(\textit{unreverse} \text{sequence})\}
\hspace{1em}⇒ \text{Return \text{sequence} in reverse order.}

\{(\textit{butlast} \text{list} \text{num})\}
\hspace{1em}⇒ \text{list excluding last num conses.}

\{(\textit{replac} \text{list} \text{cons} \text{object})\}
\hspace{1em}⇒ \text{Replace car, or cdr, respectively, of \text{cons} with \text{object}.}

\{(\textit{ldiff} \text{list} \text{foo})\}
\hspace{1em}⇒ \text{If \text{foo} is a tail of \text{list}, return preceding part of \text{list}. Otherwise return \text{list}.}

\{(\textit{adjoin} \text{foo} \text{list})\}
\hspace{1em}⇒ \text{Return \text{list} if \text{foo} is already member of \text{list}. If not, return }\text{(cons foo list)}\text{.}

\{(\textit{mpos} \text{place})\}
\hspace{1em}⇒ \text{Set place to }\text{(cadr place)}, \text{return }\text{(car place)}.\text{)

\{(\textit{mpush} \text{foo} \text{place})\}
\hspace{1em}⇒ \text{Set place to }\text{(cons foo place)}\text{.}

\{(\textit{mpushnew} \text{foo} \text{place})\}
\hspace{1em}⇒ \text{Set place to }\text{(adjoin foo place)}\text{.}

\{(\textit{append} \text{proper-list}^+ \text{for \text{seq})}\}
\hspace{1em}⇒ \text{Return concatenated list or, with only one argument, \text{foo}, \text{foo} can be of any type.}

\{(\textit{revappend} \text{list} \text{foo})\}
\hspace{1em}⇒ \text{Return concatenated list after reversing order in \text{list}.}

\{(\textit{reconc} \text{list} \text{foo})\}
\hspace{1em}⇒ \text{Return concatenated list after reversing order in \text{list}.}

\{(\textit{mapcar} \text{function list}^+)\}
\hspace{1em}⇒ \text{Return list of return values of \text{function} successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list.}

\{(\textit{map} \text{function list}^+)\}
\hspace{1em}⇒ \text{Return list of return values of \text{function} successively invoked with corresponding arguments, either cars or cdrs, respectively, from each list. \text{function} should return a list.}

\{(\textit{concat} \text{function list}^+)\}
\hspace{1em}⇒ \text{Return first list after successively applying \text{function} to corresponding arguments, either cars or cdrs, respectively, from each list. \text{function} should have some side effects.}

\{(\textit{copy-list} \text{list})\}
\hspace{1em}⇒ \text{Return copy of list with shared elements.}

4.3 Association Lists

\{(\textit{pairlis} \text{keys} \text{values} \text{alist})\}
\hspace{1em}⇒ \text{Prepend to \text{alist} an association list made from lists \text{keys} and \text{values}.}

\{(\textit{acons} \text{key} \text{value} \text{alist})\}
\hspace{1em}⇒ \text{Return \text{alist} with a (\text{key} . \text{value}) pair added.}

\{(\textit{associ} \text{foo} \text{alist})\}
\hspace{1em}⇒ \text{Return \text{alist} with a (\text{key} . \text{value}) pair added.}

\{(\textit{assoc-if-not} \text{test} \text{alist})\}
\hspace{1em}⇒ \text{First cons whose car, or cdr, respectively, satisfies \text{test}.}

\{(\textit{copy-alist} \text{alist})\}
\hspace{1em}⇒ \text{Return copy of \text{alist}.}
4.4 Trees

\[
\begin{align*}
\text{(tree-equal foo bar \{test test\})} & \quad \triangleright \text{Return } T \text{ if trees } \text{foo} \text{ and } \text{bar} \text{ have same shape and leaves satisfying test.} \\
\text{(tree-not test)} & \quad \triangleright \text{Return } T \text{ if trees } \text{foo} \text{ and } \text{bar} \text{ have same shape and leaves satisfying test.} \\
\text{(make-tree new old tree)} & \quad \triangleright \text{Make copy of tree with each subtree or leaf matching old replaced by new.} \\
\text{(make-tree new old tree)} & \quad \triangleright \text{Make copy of tree with each subtree or leaf matching old replaced by new.} \\
\text{(alist-setf association-list tree \{test function\})} & \quad \triangleright \text{Make copy of tree with each subtree or leaf matching key in association-list replaced by that key's value.} \\
\text{(copy-tree tree)} & \quad \triangleright \text{Copy of tree with same shape and leaves.}
\end{align*}
\]

4.5 Sets

\[
\begin{align*}
\text{(intersection a b)} & \quad \triangleright \text{Return } a \cap b \text{, or } a \cup b, \text{ or } a \triangle b, \text{ respectively, of lists } a \text{ and } b. \\
\text{(set-difference a b)} & \quad \triangleright \text{Return } a \setminus b. \\
\text{(union a b)} & \quad \triangleright \text{Return } a \cup b. \\
\text{(set-exclusive-or a b)} & \quad \triangleright \text{Return } a \triangle b. \\
\text{(intersection a b)} & \quad \triangleright \text{Return } a \cap b. \\
\text{(set-difference a b)} & \quad \triangleright \text{Return } a \setminus b. \\
\text{(union a b)} & \quad \triangleright \text{Return } a \cup b. \\
\text{(set-exclusive-or a b)} & \quad \triangleright \text{Return } a \triangle b. \\
\text{(intersection a b)} & \quad \triangleright \text{Return } a \cap b. \\
\text{(set-difference a b)} & \quad \triangleright \text{Return } a \setminus b. \\
\text{(union a b)} & \quad \triangleright \text{Return } a \cup b. \\
\text{(set-exclusive-or a b)} & \quad \triangleright \text{Return } a \triangle b.
\end{align*}
\]

5 Arrays

5.1 Predicates

\[
\begin{align*}
\text{(arrayp foo)} & \quad \triangleright T \text{ if } \text{foo} \text{ is of indicated type.} \\
\text{(vectorp foo)} & \quad \triangleright T \text{ if } \text{foo} \text{ is of indicated type.} \\
\text{(simple-vector-p foo)} & \quad \triangleright T \text{ if } \text{foo} \text{ is of indicated type.} \\
\text{(bit-vector-p foo)} & \quad \triangleright T \text{ if } \text{foo} \text{ is of indicated type.} \\
\text{(bit-array-p foo)} & \quad \triangleright T \text{ if } \text{foo} \text{ is of indicated type.} \\
\text{(adjustable-array-p array)} & \quad \triangleright T \text{ if array is adjustable/has a fill pointer, respectively.} \\
\text{(array-has-fill-pointer-p array)} & \quad \triangleright T \text{ if array has fill pointer.} \\
\text{(array-in-bounds-p array \{subscripts\})} & \quad \triangleright T \text{ if subscripts are in array's bounds.}
\end{align*}
\]

5.2 Array Functions

\[
\begin{align*}
\text{(make-array \{dimension-sizes\} \{element-type\} \{fill-pointer\} \{initial-element\} \{initial-contents\} \{displaced-to\} \{displaced-index-offset\})} & \quad \triangleright \text{Return fresh, or adjust, respectively, vector or array.} \\
\text{(aref array \{subscripts\})} & \quad \triangleright \text{Return array element pointed to by subscripts.} \\
\text{(row-major-aref array i)} & \quad \triangleright \text{Return } i \text{th element of array in row-major order.}
\end{align*}
\]

5.3 Vector Functions

Vectors can as well be manipulated by sequence functions; see section 6.

\[
\begin{align*}
\text{(vector foo *)} & \quad \triangleright \text{Return fresh simple vector of } \text{foo}. \\
\text{(sref vector i)} & \quad \triangleright \text{Element } i \text{ of simple vector.} \\
\text{(vector-push foo vector)} & \quad \triangleright \text{Return } \text{NIL} \text{ if vector's fill pointer equals size of vector. Otherwise replace element of vector pointed to by fill pointer with } \text{foo}, \text{ then increment fill pointer.} \\
\text{(vector-push-extend foo vector \{num\})} & \quad \triangleright \text{Replace element of vector pointed to by fill pointer with } \text{foo}, \text{ then increment fill pointer. Extend vector's size by } \geq \text{num} \text{ if necessary.} \\
\text{(vector-pop vector)} & \quad \triangleright \text{Return element of vector its fillpointer points to after decrementation.} \\
\text{(fill-pointer vector)} & \quad \triangleright \text{Fill pointer of vector.}
\end{align*}
\]
Variables

- `(defconstant symbol form doc)`
  - Assign value of form to global constant/dynamic variable `symbol`.

- `(defparameter symbol form doc)`
  - Unless bound already, assign value of form to dynamic variable `symbol`.

- `(setf symbol form)`
  - Set symbol's value cell to `form`. Deprecated.

- `(multiple-value-setq vars form)`
  - Set elements of `vars` to the values of `form`. Return `form`'s primary value.

- `(shiftf place+ foo)`
  - Store value of `foo` in rightmost `place` shifting values of `places` left, returning first `place`.

- `(_rotatef place+)`
  - Rotate values of `place` left, old first becoming new last `place`'s value. Return `NIL`.

- `(makunbound foo)`
  - Delete special variable `foo` if any.

- `(get symbol key [default nil])`
  - First entry `key` from property list stored in `symbol`/in `place`, respectively, or `default` if there is no `key`. `setf`able.

- `(get-properties property-list keys)`
  - Return key and value of first entry from `property-list` matching a key from `keys`, and tail of `property-list` starting with that key. Return `NIL`, `NIL`, and `NIL` if there was no matching key in `property-list`.

- `(remprop symbol key)`
  - Remove first entry `key` from property list stored in `symbol`/in `place`, respectively. Return `T` if `key` was there, or `NIL` otherwise.

- `(remf place key)`
  - Remove all property lists with `place`.

- `(progv symbols values forms)`
  - Evaluate `forms` with locally established dynamic bindings of `symbols` to `values` or `NIL`. Return values of `forms`.

- `(let ((name [value]) ...)) (declare (declaim *) forms)`
  - Evaluate `forms` with names lexically bound (in parallel or sequentially, respectively) to `values`. Return values of `forms`.

- `(multiple-value-bind (bar* values-form) (declare (declaim *) body-forms))`
  - Evaluate `body-forms` with `vars` lexically bound to the return values of `values-form`. Return values of `body-forms`.

- `(find position foo sequence)`
  - Return first element in `sequence` which matches `foo`, or its position relative to the begin of `sequence`, respectively.

- `(find-if position-if ...)`
  - Return first element in `sequence` which satisfies `test`, or its position relative to the begin of `sequence`, respectively.

- `(search sequence-a sequence-b)`
  - Search `sequence-b` for a subsequence matching `sequence-a`. Return position in `sequence-b`, or `NIL`.

- `(remove foo sequence)`
  - Delete `foo` sequence.

- `(remove-if test sequence)`
  - Make copy of `sequence` without elements matching `foo`.

- `(remove-if-not test sequence)`
  - Make copy of `sequence` with all (or `count`) elements satisfying `test` removed.

- `(remove-duplicates sequence)`
  - Make copy of `sequence` without duplicates.

- `(substitute old sequence)`
  - Make copy of `sequence` with all (or `count`) `olds` replaced by `new`.

- `(substitute-if old sequence)`
  - Make copy of `sequence` with all (or `count`) `olds` satisfying `test` replaced by `new`.

- `(substitute-if-not old sequence)`
  - Make copy of `sequence` with all (or `count`) `olds` not satisfying `test` replaced by `new`.

- `(substitute-if-not old sequence)`
  - Make copy of `sequence` with all (or `count`) `olds` not satisfying `test` replaced by `new`.

- `(substitute-if-not old sequence)`
  - Make copy of `sequence` with all (or `count`) `olds` not satisfying `test` replaced by `new`.
7 Hash Tables

The Loop Facility provides additional hash-table-related functionality; see loop, page 21.

Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 9 and 16.

\begin{align*}
\texttt{(hash-table-p foo)} & \quad \text{Return T if foo is of type hash-table.} \\
\texttt{(make-hash-table \{ :size int \, :rehash-size num \, :rehash-threshold num \})} & \quad \text{Make a hash table.} \\
\texttt{(gethash key hash-table [default nil])} & \quad \text{Return object with key if any or default otherwise; and T if found, NIL otherwise.} \\
\texttt{(hash-table-count hash-table)} & \quad \text{Number of entries in hash-table.} \\
\texttt{(remhash key hash-table)} & \quad \text{Remove from hash-table entry with key and return T if it existed. Return NIL otherwise.} \\
\texttt{(clrhash hash-table)} & \quad \text{Empty hash-table.} \\
\texttt{(maphash function hash-table)} & \quad \text{Iterate over hash-table calling function on key and value. Return NIL.} \\
\texttt{(*with-hash-table-iterator (foo hash-table) (declare (type-of hash-table)) form)} & \quad \text{Return values of forms. In forms, invocations of (foo) return: T if an entry is returned; its key; its value.} \\
\texttt{(hash-table-test hash-table)} & \quad \text{Test function used in hash-table.} \\
\texttt{(hash-table-size hash-table)} & \quad \text{Current size of hash-table.} \\
\texttt{(hash-table-rehash-size hash-table)} & \quad \text{Current size of hash-table.} \\
\texttt{(hash-table-rehash-threshold hash-table)} & \quad \text{Current rehash-threshold value.} \\
\texttt{(sexhash foo)} & \quad \text{Hash code unique for any argument, equal foo.}
\end{align*}

8 Structures

\begin{align*}
\texttt{(defstruct}} & \quad \text{Define structure together with functions MAKE-foo, COPY-foo and foo-P, and settable accessor foo-slot.} \\
\texttt{(copy-structure structure)} & \quad \text{Return copy of structure with shared slot values.}
\end{align*}

9 Control Structure

9.1 Predicates

\begin{align*}
\texttt{(eq foo bar)} & \quad \text{If foo and bar are identical.} \\
\texttt{(eql foo bar)} & \quad \text{If foo and bar are equal, or the same character, or numbers of the same type and value.} \\
\texttt{(equal foo bar)} & \quad \text{If foo and bar are equal, or are equivalent pathnames, or are conses with equal cars and cdrs, or are strings or bit-vectors with equal elements below their fill pointers.} \\
\texttt{(equalp foo bar)} & \quad \text{If foo and bar are identical; or are the same character ignoring case; or are numbers of the same value ignoring type; or are equivalent pathnames; or are conses or arrays of the same shape with equal elements; or are structures of the same type with equal elements; or are hash-tables of the same size with the same :test function, the same keys in terms of :test function, and equal elements.}
\end{align*}

\begin{align*}
\texttt{(not foo)} & \quad \text{T if foo is NIL; NIL otherwise.} \\
\texttt{(boundp symbol)} & \quad \text{T if symbol is a special variable.} \\
\texttt{(constantp foo \{environment\} \{tag\})} & \quad \text{T if foo is a constant form.} \\
\texttt{(functionp foo)} & \quad \text{T if foo is of type function.}
\end{align*}
\(\text{case}\) \text{test} \left\{ \begin{array}{l}
\text{\{key\} \to foo^*} \\
\text{\{otherwise\} \to \{bar\}}
\end{array} \right\}

- Return the values of the first \(foo^*\) one of whose keys is \text{eq}. Return values of \(bar\) if there is no matching key.

\(\text{and}\) \text{form} \textit{m}

- Evaluate forms from left to right. Immediately return \text{NIL} if one form's value is \text{NIL}. Return values of last form otherwise.

\(\text{or}\) \text{form} \textit{n}

- Evaluate forms from left to right. Immediately return primary value of first non-\text{NIL}-evaluating form, or all values if last form is reached. Return \text{NIL} if no form returns \text{T}.

\(\text{progn}\) \text{form} \textit{m}

- Evaluate forms sequentially. Return values of last form.

\(\text{multiple-value-prog1}\) \textit{form-r} \text{form} \textit{m}

- Evaluate forms in order. Return values/primary value, respectively, of form-r.

\(\text{map}\) \text{progn} \textit{m}

- Evaluate \text{tagbody} like body with names lexically bound (in parallel or sequentially, respectively) to values. Return \text{NIL} or explicitly returned values. Implicitly, the whole form is a \text{block} named \text{NIL}.

\(\text{unwind-protect}\) \text{progn} \textit{m}

- Evaluate \text{protected} and then, no matter how control leaves \text{protected}, \text{cleanup}. Return values of \text{protected}.

\(\text{block}\) \text{name} \textit{m}

- Evaluate forms in a lexical environment, and return their values unless interrupted by \text{return-from}.

\text{return-from} \text{foo} \text{result} \textit{m}

- Have nearest enclosing \text{block} named \text{foo} named \text{NIL}, respectively, return with values of result.

\(\text{tagbody}\) \text{tag} \textit{m}

- Evaluate forms in a lexical environment. \text{tags} (symbols or integers) have lexical scope and dynamic extent, and are targets for \text{go}. Return \text{NIL}.

\(\text{go}\) \textit{tag}

- Within the innermost possible enclosing \text{tagbody} jump to a tag, \text{eq} tag.

\(\text{catch}\) \text{tag} \textit{m}

- Evaluate forms and return their values unless interrupted by \text{throw}.

\(\text{throw}\) \text{tag} \textit{m}

- Have the nearest dynamically enclosing \text{catch} with a tag \text{eq} return with the values of \text{form}.

\(\text{sleep}\) \text{n}

- Wait \text{n} seconds; return \text{NIL}.

\(\text{return}\) \text{form} \textit{m}

- Evaluate forms with variables from tree \text{destruct-lambda} bound to corresponding elements of tree \text{bar}, and return their values. \text{destruct-lambda} resembles \text{macro-lambda} (section 9.4), but without any \text{environment} clause.

\text{defun} \text{return} \text{form} \textit{m}

- Define a function named \text{return} or \text{setf}, or an anonymous function, respectively, which applies forms to \text{ord-ls}. For \text{defun}, forms are enclosed in an implicit \text{block} named \text{return}.

\(\text{let}\) \text{labels} \textit{m}

- Evaluate forms with locally defined functions \text{return}. Globally defined functions of the same name are shadowed. Each \text{return} also the name of an implicit \text{block} around its corresponding \text{local-form*}. Only for \text{labels}, functions \text{return} are visible inside \text{local-forms}. Return values of forms.

\(\text{function}\) \text{return} \text{form} \textit{m}

- Return lexically innermost function named \text{return} or a lexical closure of the \text{lambda} expression.

\text{apply} \text{function} \text{arg} \textit{m}

- Values of \text{return} called with \text{arg} and the list elements of \text{arg}. \text{setf}able if function is one of \text{aref}, \text{bit}, and \text{sbit}.

\text{funcall} \text{function} \text{arg} \textit{m}

- Values of \text{return} called with \text{arg}.

\(\text{multiple-value-call}\) \text{function} \text{form} \textit{m}

- Call \text{return} with all the values of each \text{form} as its arguments. Return values returned by \text{return}.

\text{values-list} \text{list} \textit{m}

- Return elements of \text{list}.

\text{values} \text{foo} \textit{m}

- Return as multiple values the primary values of the \text{foo}s. \text{setf}able.

\(\text{multiple-value-list}\) \text{form} \textit{m}

- List of the values of \text{form}.

\text{nth-value} \text{n} \text{form} \textit{m}

- Zero-indexed \text{n}th return value of \text{form}.

\text{complement} \text{function} \textit{m}

- Return new function with same arguments and same side effects as \text{return}, but with complementary truth value.

\text{constantly} \text{function} \textit{m}

- Function of any number of arguments returning \text{foo}.

\text{identity} \text{foo} \textit{m}

- Return \text{foo}.
9.4 Macros

Below, macro lambda list (macro-λ*) has the form of either

\[
[(\&whole \text{var} [E] (macro-λ*) (E)] \&optional \text{var} \&rest \text{var} \&body \text{var} \&key \text{var} \&allow-other-keys \text{var} \&environment \text{var}]
\]

or

\[
[(\&whole \text{var} [E] (macro-λ*) (E) \&optional \text{var} \&body \text{var} \&key \text{var} \&allow-other-keys \text{var} \&environment \text{var}]
\]

One toplevel [E] may be replaced by \&environment \text{var}. \text{supplied-p} is \text{T} if there is a corresponding argument. \text{init} forms can refer to any \text{init} and \text{supplied-p} to their left.

\[
(\text{makedefmacro} \text{foo form})
\]

\[
(\text{makedefsymbolmacro} \text{foo form})
\]

\[
(\text{makemacro} \text{foo (lambda-form)})
\]

\[
(\text{makedefsetf function updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})
\]

\[
(\text{makedefsetf function updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})
\]

9.5 Control Flow

\[
(\text{if test then \text{elsevar}})
\]

\[
(\text{cond test \text{body})}
\]

\[
(\text{when \text{unless} \text{test doc}})
\]

\[
(\text{defsetf function updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})
\]

\[
(\text{makedefsetf function updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})
\]

\[
(\text{makedefmacro foo \text{macro-form}})
\]

\[
(\text{makedefsymbolmacro \text{foo form}})
\]

\[
(\text{makemacro \text{foo (lambda-form)}})
\]

\[
(\text{makedefsetf function \text{updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})}
\]

\[
(\text{makedefmacro \text{foo \&macro-form})}
\]

\[
(\text{makedefsymbolmacro \text{foo form})}
\]

\[
(\text{makemacro \text{foo (lambda-form)})}
\]

\[
(\text{makedefsetf function \text{updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})}
\]

\[
(\text{makedefmacro \text{foo \&macro-form})}
\]

\[
(\text{makedefsymbolmacro \text{foo form})}
\]

\[
(\text{makemacro \text{foo (lambda-form)})}
\]

\[
(\text{makedefsetf function \text{updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})}
\]

\[
(\text{makedefmacro \text{foo \&macro-form})}
\]

\[
(\text{makedefsymbolmacro \text{foo form})}
\]

\[
(\text{makemacro \text{foo (lambda-form)})}
\]

\[
(\text{makedefsetf function \text{updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})}
\]

\[
(\text{makedefmacro \text{foo \&macro-form})}
\]

\[
(\text{makedefsymbolmacro \text{foo form})}
\]

\[
(\text{makemacro \text{foo (lambda-form)})}
\]

\[
(\text{makedefsetf function \text{updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})}
\]

\[
(\text{makedefmacro \text{foo \&macro-form})}
\]

\[
(\text{makedefsymbolmacro \text{foo form})}
\]

\[
(\text{makemacro \text{foo (lambda-form)})}
\]

\[
(\text{makedefsetf function \text{updater doc \lambda-var \lambda-body \lambda-keys \lambda-environment})}
\]
10 CLOS

10.1 Classes

(defun class-of (foo) ...)  
▷ Class foo is a direct instance of.

(defun change-class (new-class) ...)  
▷ Change class of instance to new-class. Retain the status of any slots that are common between instance’s original class and new-class. Initialize any newly added slots with the values of the corresponding initargs if any, or with the values of their initform forms if not.

(defun make-instances-obsolete)  
▷ Update all existing instances of class using update-instance-for-redefined-class.

(defun initialize-instance ...)  
▷ Set slots on behalf of make-instance of change-class by means of _shared-initialize.

(defun make-instance class ...)  
▷ Make new instance of class.

(defun reinitialize-instance instance ...)  
▷ Change local slots of instance according to initargs by means of _shared-initialize.

(defun slot-value foo slot)  
▷ Return value of slot in foo, if accessible.

(defun slot-unbound instance slot)  
▷ Make slot in instance unbound.

(defun with-slots ((slot var slot) ...) instance)  
▷ Return values of forms after evaluating them in a lexical environment with slots of instance visible as setfable slots or vars with accessors of instance visible as setfable vars.

(defun with-accessors ((var accessor) ...) form)  
▷ Return values of forms after evaluating them in a lexical environment with slots of instance visible as setfable slots or vars with accessors of instance visible as setfable vars.

(defun find-class symbol)  
▷ Return class named symbol, if setfable.

(defun make-instance class ...)  
▷ Make new instance of class.

(defun reinitialize-instance instance ...)  
▷ Change local slots of instance according to initargs by means of _shared-initialize.

(defun slot-value foo slot)  
▷ Return value of slot in foo, if accessible.

(defun slot-unbound instance slot)  
▷ Make slot in instance unbound.

(defun with-slots ((slot var slot) ...) instance)  
▷ Return values of forms after evaluating them in a lexical environment with slots of instance visible as setfable slots or vars with accessors of instance visible as setfable vars.

(defun with-accessors ((var accessor) ...) form)  
▷ Return values of forms after evaluating them in a lexical environment with slots of instance visible as setfable slots or vars with accessors of instance visible as setfable vars.

(defun find-class symbol)  
▷ Return class named symbol, if setfable.

9.6 Iteration

(defun do form ...)  
▷ Evaluates tagbody-like body with vars successively bound to values of the corresponding start and step forms. Vars are bound in parallel/sequentially, respectively. Stop iteration when stop is T. Return values of result. Implicitly, the whole form is a _block named NIL.

(defun do-mtimes (var init result-limit) form)  
▷ Evaluates tagbody-like body with vars successively bound to integers from 0 to i - 1. Upon evaluation of result, var is i. Implicitly, the whole form is a _block named NIL.

(defun do-dolist (var list result-limit) form)  
▷ Evaluates tagbody-like body with vars successively bound to the elements of list. Upon evaluation of result, var is NIL. Implicitly, the whole form is a _block named NIL.

9.7 Loop Facility

(defun loop ...)  
▷ Simple Loop. If forms do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit _block named NIL.

(defun loop *)  
▷ Loop Facility. For Loop Facility keywords see below and Figure 1.

(defun named ...)  
▷ Give _loop’s implicit _block a name.

(defun with ...)  
▷ Begin destructuring type specifier _d-type as with.

(defun loop *)  
▷ Loop Facility. For Loop Facility keywords see below and Figure 1.
Figure 1: Loop Facility, Overview.
(make-condition condition-type \{binary-name\} \*form\*)
> Return new instance of condition-type.

(signal condition \{binary-name\} \{control arg\} \*form\*)
> Unless handled, signal as condition, warning or error, respectively, condition or a new instance of condition-type or, with \{format\} control and args (see page 36), simple-condition, simple-warning, or simple-error, respectively. From signal and warn, return NIL.

(error continue-control condition \{binary-name\} \{control arg\} \*form\*)
> Unless handled, signal as correctable error condition or a new instance of condition-type or, with \{format\} control and args (see page 36), simple-error. In the debugger, use \{format\} arguments continue-control and continue-args to tag the continue option. Return NIL.

(ignore-errors \*form\*)
> Return values of forms or, in case of errors, NIL and the condition.

:invoke-debugger condition
> Invoke debugger with condition.

(assert test \{place\} \{condition \{binary-name\} \{control arg\} \*form\*)
> If test, which may depend on places, returns NIL signal as correctable error condition or a new instance of condition-type or, with \{format\} control and args (see page 36), error. When using the debugger's continue option, places can be altered before re-evaluation of test. Return NIL.

(handler-case foo \{type \{var\} \{declare \{decl\}\} \{condition \{binary-name\} \{control arg\} \*form\*)
> If, on evaluation of foo, a condition of type is signalled, evaluate matching \{condition\}-forms with var bound to the condition, and return their values. Without a condition, bind var to values of \{condition\} or return values of forms or, without a \{no-error\} clause, return values of foo. See page 17 for \{control\}.

(handler-bind \{condition-type \{handler-function\} \*form\*)
> Return values of forms after evaluating them with condition-types dynamically bound to their respective \{handler-function\}-functions of argument \{condition\}.

(with-simple-restart \{restart \{NIL\} \{control arg\} \*form\*)
> Return values of forms unless \{restart\} is called during their evaluation. In this case, describe \{restart\} using \{format\} control and args (see page 36) and return NIL and T.

(restart-case form \{restart \{ord-lambda\} \*form\*)
> Return values of form or, if during evaluation of form one of the dynamically established \{restart\}'s is called, the values of its \{restart\}-forms. A \{restart\} is visible under condition if \#funcall \#\{test-function \{condition\}\} returns T. If presented in the debugger, \{restart\}'s are described by \#\{string\} or by \#\{report-function\} (of a stream). A \{restart\} can be called by \#\{invoke-restart \{restart \{ord-lambda\} \} \*form\*) where a list of the respective args is supplied by \#\{report-function\}. See page 17 for \{ord-lambda\}.

(update-instance-for-redefined-class new-instance \{added-slots\} \{discarded-slots\} \{discarded-slots\}-property-list \{binary-name\} \*other-kegarg\*)
> On behalf of \#\{make-\{instances\}-obsolete\} and by means of \#\{shared-initialize\}, set any \{instance\} slots to their corresponding values; set any remaining \{added-slots\} to the values of their \{instance\}-forms. Not to be called by user.

(allocate-instance-class \{binary-name\} \*other-kegarg\*)
> Return uninitialized instance of class. Called by \#\{make-instance\}.

(slot-missing class instance slot \{setf \{slot\}-boundp \{slot\}-makunbound \{slot\}-value \*value\})
> Called on attempted access to non-existing or unbound slot. Default methods signal error/unbound-slot, respectively. Not to be called by user.

10.2 Generic Functions

(next-method-p)
> T if enclosing method has a next method.

(defgeneric foo \{setf foo\} \{required-var\} \{&optional \{var\}\} \{&allow-other-keys\})
> Define or modify generic function \{foo\}. Remove any methods previously defined by defgeneric. \{gf-class\} and the lambda parameters \{required-var\} and \{var\} must be compatible with existing methods. defmethod-args resemble those of defmethod. For \{c-type\} see section 10.3.

(ensure-generic-function foo \{setf foo\})
> Define or modify generic function \{foo\}. \{gf-class\} and the lambda parameters \{required-var\} and \{var\} must be compatible with existing methods. defmethod-args must be set. defmethod-args resemble those of defmethod. For \{c-type\} see section 10.3.
10.3 Method Combination Types

- **standard**
  - Evaluate most specific `around` method supplying the values of the generic function. From within this method, `call-next-method` can call less specific `around` methods if there are any. If not, or if there are no `around` methods at all, call all `before` methods, most specific first, and the most specific primary method which supplies the values of the calling `call-next-method` if any, or of the generic function; and which can call less specific primary methods via `call-next-method`. After its return, call all `after` methods, least specific first.

- **appendlist|concat|progn|max|min**
  - Simple, built-in `method-combination` types; have the same usage as the `c-type` defined by the short form of `define-method-combination`.

- **define-method-combination**
  - Define new method for generic function `foo`. `spec-vars` specialize to either being of class or being `eq` `bar`, respectively. On invocation, `vars` and `spec-vars` of the new method act like parameters of a function with `body` form. `forms` are enclosed in an implicit `block` `foo`. Applicable `qualifiers` depend on the `method-combination` type; see section 10.3.

- **add-method**
  - Add (if necessary) or remove (if any) method to/from `generic-function`.

- **find-method**
  - Return `method` or `error`.

- **compute-applicable-methods**
  - List of methods suitable for `args`, most specific first.

- **call-next-method**
  - From within a method, call next method with `args`; return its values.

- **no-applicable-method**
  - Called on invocation of `generic-function` on `args` if there is no applicable method. Default method signals `error`. Not to be called by user.

- **invalid-method-error**
  - Signal `error` on applicable method with invalid `qualifiers`, or on method combination. For `control` and `args` see `format`, page 36.

- **next-method**
  - Called on invocation of `call-next-method` when there is no next method. Default method signals `error`. Not to be called by user.

- **function-keywords**
  - Return list of keyword parameters of `method` and `T` if other keys are allowed.

- **method-qualifiers**
  - List of `qualifiers` of `method`.

### 11 Conditions and Errors

**Define condition foo**

```lisp
(define-condition foo (parent-type :common) (slotted ()
  (slot :reader reader)
  (slot :writer (setf writer))
  (slot :accessor accessor)
  (slot :allocation instance class)
  (slot :initial-form form)
  (slot :documentation slot-doc)
  (slot :default-initargs (name value))
  (slot :report (string report-function)))
```

- Define, as a subtype of `parent-types`, condition `foo`. In a new condition, a `slot`'s `value` defaults to `form` unless set via `:initarg`; it is readable via `reader` or `accessor`, and writable via `writer` or `setter` `accessor` value. With `:allocation` `class`, `slot` is shared by all conditions of `type` `foo`. A condition is reported by `string` or by `report-function` of arguments condition and stream.

**Short Form.** Define new `method-combination` `c-type`. From within this method, `call-next-method` can call less specific `around` methods if there are any. If not, or if there are no `around` methods at all, return from the calling `call-next-method` or from the primary method, respectively, the values of `(operator` `primary-method` gen-arg`∗`), `gen-arg`∗` being the arguments of the generic function. The `primary-methods` are ordered `[^most-specific-first]` `[^most-specific-last]` (specified as `c-type` in `mdefgeneric`). Using `c-type` as the `qualifier` in `mdefmethod` makes the method primary.
13 Input/Output

13.1 Predicates

(=streamp foo) >> T if foo is of indicated type.

(=pathnamep foo) >> T if foo is of indicated type.

(=readablep foo) >> T if foo is of indicated type.

(=input-stream-p stream) >> Return T if stream is for input, for output, interactive, or open, respectively.

(=output-stream-p stream) >> Return T if stream is for input, for output, interactive, or open, respectively.

(=interactive-stream-p stream) >> Return T if stream is for input, for output, interactive, or open, respectively.

(=open-stream-p stream) >> Return T if stream is for input, for output, interactive, or open, respectively.

(=pathname-match-p path wildcard) >> T if path matches wildcard.

(=wild-pathname-p path [[:host]:device]:directory]:name]:type] version[NIL]) >> Return T if indicated component in path is wildcard. (NIL indicates any component.)

13.2 Reader

(=führer-p control arg) >> Ask user a question and return T or NIL depending on their answer. See page 36, format, for control and args.

(=yes-or-p) >> Evaluate forms with standard behaviour of reader and printer. Return values of forms.

(=read-preserving-whitespace stream [standard-input] [eof-error] [eof-value [recursive NIL]]) >> Read printed representation of object.

(=read-from-string string [eof-error] [eof-value [start start] [end end] [preserve-whitespace bool]]) >> Return object read from string and zero-indexed position of next character.

(=read-delimited-list char stream [standard-input] [recursive NIL]) >> Continue reading until encountering char. Return list of objects read. Signal error if no char is found in stream.

(=read-char stream [standard-input] [eof-error] [eof-value [recursive NIL]]) >> Return next character from stream.

(=read-char-no-hang stream [standard-input] [eof-error] [eof-value [recursive NIL]]) >> Next character from stream or NIL if none is available.

(=peek-char mode stream [standard-input] [eof-error] [eof-value [recursive NIL]]) >> Next, or if mode is 1, next non-whitespace character, or if mode is a character, next instance of it, from stream without removing it there.

(=unread-char character stream [standard-input]) >> Put last read character back into stream; return NIL.

(=read-byte stream [eof-error] [eof-value NIL]) >> Read next byte from binary stream.

(=read-line stream [standard-input] [eof-error] [eof-value [recursive NIL]]) >> Return a line of text from stream and T if line has been ended by end of file.
12 Types and Classes

For any class, there is always a corresponding type of the same name.

\( \text{typep} \ foo \ \text{[environment]} \) ▷ \( T \) if \( \text{foo} \) is of type.

\( \text{subtypep} \ \text{type-a} \ \text{type-b} \ \text{[environment]} \)
▷ Return \( T \) if \( \text{type-a} \) is a recognizable subtype of \( \text{type-b} \), and \( NIL \) if the relationship could not be determined.

\( \text{the} \ \text{type form} \) ▷ Declare values of \text{form} to be of type.

\( \text{coerce} \ \text{object type} \) ▷ Coerce \text{object} into type.

\( \text{etypecase} \ \text{foo} \ \{ \ \text{otherwise} \ \text{b-form} \} \)
▷ Return values of the first \text{a-form} whose type is \text{foo} of. Return values of \text{b-forms} if no type matches.

\( \text{etypecase} \ \text{foo} \ \{ \text{type form} \} \)
▷ Return values of the first \text{form} whose type is \text{foo} of. Signal non-correctable/correctable \text{type-error} if no type matches.

\( \text{type-of} \ \text{foo} \)
▷ Type of \text{foo}.

\( \text{check-type} \ \text{place type [string \ b-form]} \)
▷ Signal correctable \text{type-error} if \text{place} is not of type. Return \( NIL \).

\( \langle \text{stream-element-type} \ \text{stream} \rangle \)
▷ Type of stream objects.

\( \langle \text{array-element-type} \ \text{array} \rangle \)
▷ Element \text{type} \text{array} can hold.

\( \langle \text{upgraded-array-element-type} \ \text{type [environment]} \rangle \)
▷ Element \text{type} of most specialized array capable of holding elements of \text{type}.

\( \langle \text{deftype} \ \text{foo (macro-λ*) (declare doc*) \ [doc form*]} \rangle \)
▷ Define type \text{foo} which when referenced as \text{foo} \( \text{arg*} \) (or as \text{foo} if \text{macro-λ} doesn’t contain any required parameters) applies expanded \text{forms} to \text{args} returning the new type. For \( \text{macro-λ*} \) see page 18 but with default value of \( * \) instead of \( NIL \). \text{forms} are enclosed in an implicit \text{block} named \text{foo}.

\( \text{eq} \ \text{foo} \)
\( \text{member} \ \text{foo*} \)
▷ Specifier for a type comprising \text{foo} or \text{foo}s.

\( \text{satisfies} \ \text{predicate} \)
▷ Type specifier for all objects satisfying \text{predicate}.

\( \text{mod} \ \text{n} \)
▷ Type specifier for all non-negative integers \( < \) \text{n}.

\( \text{not} \ \text{type} \)
▷ Complement of \text{type}.

\( \text{and} \ \text{type*} \)
▷ Type specifier for intersection of \text{types}.

\( \text{or} \ \text{type*} \)
▷ Type specifier for union of \text{types}.

\( \langle \text{values} \ \text{type*} \ \{ \langle \text{optional} \ \text{type*} \ \langle \text{rest} \ \text{other-args} \rangle \} \rangle \)
▷ Type specifier for multiple \text{values}.

\( \ast \)
▷ As a \text{type} argument (cf. Figure 2): no restriction.

Figure 2: Precedence Order of System Classes ( ), Classes ( ), Types ( ), and Condition Types ( ). Every type is also a supertype of \( NIL \), the empty type.
\[\text{(print-newline \text{linear} \text{fill} \text{miser} \text{mandatory})}\]

\(\quad\)Print a conditional newline if \text{stream} is a pretty printing stream. Return \text{NIL}.

\(\text{*print-array}\)

\(\quad\)If \text{T}, print arrays \text{readably}.

\(\text{*print-base}\)

\(\quad\)Radix for printing rationals, from 2 to 36.

\(\text{*print-case}\)

\(\quad\)Print symbol names all uppercase (\text{:upcase}), all lowercase (\text{:downcase}), capitalized (\text{:capitalize}).

\(\text{*print-circle}\)

\(\quad\)If \text{T}, avoid indefinite recursion while printing circular structure.

\(\text{*print-escape}\)

\(\quad\)If \text{NIL}, do not print escape characters and package prefixes.

\(\text{*print-gensym}\)

\(\quad\)If \text{T}, print \#: before uninterned symbols.

\(\text{*print-length}\)

\(\quad\)If \text{T}, restrict printing of objects to that number of elements per level/10 that depth/10 that number of lines.

\(\text{*print-miser-width}\)

\(\quad\)If integer and greater than the width available for printing a substructure, switch to the more compact miser style.

\(\text{*print-pretty}\)

\(\quad\)If \text{T}, print prettily.

\(\text{*print-radix}\)

\(\quad\)If \text{T}, print rationals with a radix indicator.

\(\text{*print-readably}\)

\(\quad\)If \text{T}, print \text{readably} or signal error \text{print-not-readable}.

\(\text{*print-right-margin}\)

\(\quad\)Right margin width in ems while pretty-printing.
Common Lisp Quick Reference

\[ n/d \quad \Rightarrow \quad \text{The ratio } \frac{n}{d} \]

\[
\{ [m.n] [(S|F|I|L|E)] \} \quad \Rightarrow \quad m.n \cdot 10^n \quad \text{as short-float, single-float, double-float,}
\]

\[ \text{long-float, or the type from \texttt{read-default-format}.} \]

\#C\( a \ b \) \quad \Rightarrow \quad \text{\( (\# \text{complex} \ a \ b) \), the complex number } a + bi. \]

\#\'\( \text{foo} \) \quad \Rightarrow \quad \text{\( (\text{function} \ \text{foo}) \), the function named } \text{foo}. \]

\#\n\text{sequence} \quad \Rightarrow \quad n\text{-dimensional array.} \]

\#\[n\]/\text{foo}\* \quad \Rightarrow \quad \text{Vector of some (or } n) \text{ foos filled with last } \text{foo} \text{ if necessary.} \]

\[#\n\text{b}\* \quad \Rightarrow \quad \text{Bit vector of some (or } n) \text{ bs filled with last } b \text{ if necessary.} \]

\#\S(t\text{ype} \{\text{slot value}\}) \quad \Rightarrow \quad \text{Structure of type.} \]

\#P\text{string} \quad \Rightarrow \quad \text{A pathname.} \]

\#:\text{foo} \quad \Rightarrow \quad \text{Uninterned symbol } \text{foo}. \]

\#:\text{form} \quad \Rightarrow \quad \text{Read-time value of } \text{form}. \]

\+.\text{read-eval}\-\text{print} \quad \Rightarrow \quad \text{If } \text{NIL}, \text{ a reader-error is signalled at \#-.} \]

\#\text{integer}\=\text{foo} \quad \Rightarrow \quad \text{Give } \text{foo} \text{ the label integer.} \]

\#\text{integer}\# \quad \Rightarrow \quad \text{Object labelled integer.} \]

\#\< \quad \Rightarrow \quad \text{Have the reader signal reader-error.} \]

\#\+\text{feature} \text{when}\-\text{feature} \quad \Rightarrow \quad \text{Means when-} \text{feature if feature is } T; \text{ means unless-} \text{feature if feature is } \text{NIL}. \text{ Feature is a symbol from \{\text{features}, \text{ or \{\{\text{and}\} or} \text{\{or} \text{\} feature}\text{\}}\text{, or \{not feature\}.} \]

\{\text{features}\} \quad \Rightarrow \quad \text{List of symbols denoting implementation-dependent features.} \]

\{c\} \text{\{c\}} \quad \Rightarrow \quad \text{Treat arbitrary character(s) } c \text{ as alphabetic preserving case.} \]

13.4 Printer

\{\text{prin}\} \text{\( \Rightarrow \quad \text{Print } \text{foo to stream readable, readable between a newline and a space, readable after a newline, or human-readable without any extra characters, respectively. } \text{prin1}, \text{prin} \text{ and prin return foos.} \]

\{\text{prin1}\} \text{\( \Rightarrow \quad \text{Print } \text{foo to string readable or human-readable, respectively.} \]

\{\text{prin1-to-string}\} \text{\( \Rightarrow \quad \text{Print } \text{foo to string readable or human-readable, respectively.} \]

\{\text{print-object}\} \text{\( \Rightarrow \quad \text{Print object to stream. Called by the Lisp printer.} \]

\{\text{print-unreadable-object}\} \text{\( \Rightarrow \quad \text{Enclosed in } \#< \text{ and } \#, \text{ print foos by means of forms to stream. Return } \text{NIL.} \]

\{\text{terpri}\} \text{\( \Rightarrow \quad \text{Output a newline to stream. Return } \text{NIL.} \]

\{\text{fresh-line}\} \text{\( \Rightarrow \quad \text{Output a newline to stream and return } T \text{ unless stream is already at the start of a line.} \]

\{\text{write-char}\} \text{\( \Rightarrow \quad \text{Output char to stream.} \]

\{\text{write-string}\} \text{\( \Rightarrow \quad \text{Write string to stream without/with a trailing newline.} \]

\{\text{write-byte}\} \text{\( \Rightarrow \quad \text{Write byte to binary stream.} \]

\{\text{write-sequence}\} \text{\( \Rightarrow \quad \text{Write elements of sequence to binary or character stream.} \]

\{\text{array}\}\text{\{bool\}} \quad \Rightarrow \quad \text{base radix} \\text{\{upcase\}} \\text{\{case\}} \\text{\{downcase\}} \\text{\{capitalize\}} \\text{\{circle\}} \\text{\{escape\}} \\text{\{gensym\}} \\text{\{length\}} \\text{\{level\}} \\text{\{lines\}} \\text{\{miser-width\}} \\text{\{pprint-dispatch\}} \\text{\{pprint-discard\}} \\text{\{pretend\}} \\text{\{pretty\}} \\text{\{readably\}} \\text{\{right-margin\}} \\text{\{standard-output\}} \]

\{\text{pprint-fill}\} \text{\( \Rightarrow \quad \text{Print foos to stream and return foos, or print foos into string, respectively, after dynamically setting printer variables corresponding to keyword parameters } \{\text{print-bar\} becoming } \text{bar}. \text{ (stream keyword with } \text{write only.)} \]

\{\text{pprint-tabular}\} \text{\( \Rightarrow \quad \text{Print foos to stream. If foos is a list, print as many elements per line as possible; do the same in a table with a column width of } n \text{ ens; or print either all elements on one line or each on its own line, respectively. Return } \text{NIL.} \text{ Usable with } \text{/format directive } \text{-/}.} \]

\{\text{pprint-logical-block}\} \text{\( \Rightarrow \quad \text{Evaluate forms, which should print list, with stream locally bound to a pretty printing stream which outputs to the original stream. If list is in fact not a list, it is printed by } \text{write}. \text{ Return } \text{NIL.}} \]

\{\text{pprint-pop}\} \quad \Rightarrow \quad \text{Take next element off list. If there is no remaining tail of list, or } \text{\{pprint-lengths\} or } \text{\{pprint-circles\} indicate printing should end, send element together with an appropriate indicator to stream.} \]

\{\text{stream}\}\text{\{line\}} \text{\{line-relative\}} \text{\{section\}} \text{\{section-relative\}} \quad \Rightarrow \quad \text{Move cursor forward to column number } c + ki, k \geq 0 \text{ being as small as possible.} \]

\{\text{stream}\}\text{\{line\}} \quad \Rightarrow \quad \text{Specify indentation for innermost logical block relative to leftmost position/current position. Return } \text{NIL otherwise.} \]
13.7 Pathnames and Files

(pathname-host
  /pathname-device
  /pathname-directory
  /pathname-name
  /pathname-type
  /pathname-version
          path-or-stream)
  
  Return pathname component.

(make-pathname
  [host [host NIL unspecific]]
  [device [device NIL unspecific]]
  [directory [directory NIL unspecific]]
  [directory [directory [absolute [absolute [wild [wild [wild [wild [wild [wild [wild [wild [wild-inferior up back]]]]]]]]]]]]
  [name [name [file-name [file-name [file-name NIL unspecific]]]]]
  [type [type [file-type [wild [wild [wild [wild [wild [wild [wild [wild [wild unspecific]]]]]]]]]]]
  [version [version [newest [version [wild [wild [wild [wild [wild [wild [wild [wild [wild unspecific]]]]]]]]]]]]]]
  [valid]
  
  Construct a logical pathname if there is a logical
  pathname translation for host, otherwise construct a physical
  pathname. For :case :local, leave case of components
  unchanged. For :case :common, leave mixed-case components
  unchanged; convert all-uppercase components into local
  customary case; do the opposite with all-lowercase components.

(close stream [abort body])
  
  Close stream. Return T if stream had been open. If :abort
  is T, delete associated file.

(with-open-file (stream path open-arg) (declare decl) form)
  
  Use :open with open-args to temporarily create stream
  to path; return values of forms.

(with-open-stream (foo stream) (declare decl) form)
  
  Evaluate forms with foo locally bound to stream. Return
  values of forms.

(with-input-from-string (foo stream) (declare decl) form)
  
  Evaluate forms with foo locally bound to input
  string-stream from string. Return values of forms; store
  next reading position into index.

(with-output-to-string (foo stream) (declare decl) form)
  
  Evaluate forms with foo locally bound to an output
  string-stream. Append output to string and return values
  of forms if string is given. Return string containing output
  otherwise.

(stream-exteral-format stream)
  
  External file format designator.

*terminal-io
  
  Bidirectional stream to user terminal.

*standard-input

*standard-output

*error-output
  
  Standard input stream, standard output stream, or stan-
  dard error output stream, respectively.

*debug-io
  
  -query-io
  
  Bidirectional streams for debugging and user interaction.

minus-col [min-col [col-min [min-pad [pad-char]]]]
  
  Aesthetic/Standard. Print argument of any type for
  consumption by humans/by the reader, respectively.
  With , print NIL as () rather than nil; with @, add
  pad-chars on the left rather than on the right.

-radius [radius [pad-char [comma-char]]
          [comma-interval]]]
  
  Radix. (With one or more prefix arguments.)
  Print argument as number; with ;, group digits
  comma-interval each; with @, always prepend a sign.

-{R}-{R}-{R}-{R}-{R}-

Roman. Take argument as number and print it as
English cardinal number, as English ordinal number, as
Roman numeral, or as old Roman numeral, respectively.

-width [width [pad-char [comma-char]]
          [comma-interval]]]
  
  Decimal/Binary/Octal/Hexadecimal. Print integer
  argument as number. With , print digits
  comma-interval each; with @, always prepend a sign.

-fixed [width [dec-digits [shift [overflow-char]]
          [pad-char]]]]]
  
  Fixed-Format Floating-Point. With @, always pre-
  pend a sign.

-floating [dec-digits [exp-digits [scale-factor]
           [exp-char [pad-char]]]]]
  
  Exponential/General Floating-Point. Print argument
  as floating-point number with dec-digits after decimal
  point and exp-digits in the signed exponent. With , choose
  either -E or -F. With @, always prepend a sign.

-monetary [int-digits [width [pad-char]]]
  
  Monetary Floating-Point. Print argument as fixed-
  format floating-point number. With , put sign before
  any padding; with @, always prepend a sign.

{-C}-{C}-{C}-{C}-{C}-

Character. Print, spell out, print in #\ syntax, or
tell how to type, respectively, argument as (possibly
non-printing) character.
Common Lisp Quick Reference

13.6 Streams

- make-concatenated-stream input-stream* ]
- make-broadcast-stream output-stream* ]
- make-two-way-stream input-stream-part output-stream-part ]
- make-echo-stream from-input-stream to-output-stream ]
- make-synonym-stream variable-bound-to-stream ]

- return stream of indicated type.
- return a string-stream supplying the characters from string.
- return a string-stream accepting characters (available via /get-output-stream-string/).
- return list of streams concatenated-stream still has to read from/broadcast-stream is broadcasting to.
- return source stream or sink stream of two-way-stream/echo-stream, respectively.
- return symbol of synonym-stream.
- clear and return as a string characters on string-stream.
- return position within stream, or set it to position and return T on success.
- length foo would have in stream.
- clear input stream, return NIL
- clear output
- force-output
- finish-output

Common Lisp Quick Reference
14.4 Standard Packages

common-lisp

➤ Exports the defined names of Common Lisp except for those in the keyword package.

common-lisp-user cl-user

➤ Current package after startup; uses package common-lisp.

keyword

➤ Contains symbols which are defined to be of type keyword.

15 Compiler

15.1 Predicates

parity-p foo

➤ T if foo is a special operator.

(parity-function-p foo)

➤ T if foo is of type compiled-function.

15.2 Compilation

compile

➤ Returns compiled function or replace name’s function definition with the compiled function. Return T in case of warnings or errors, and T in case of warnings or errors excluding style-warnings.

(compiled-file file)

➤ Write compiled contents of file to out-path. Return true output path or NIL, T in case of warnings or errors, T in case of warnings or errors excluding style-warnings.

(compiled-file- pathname file [output-file path] [other-keyargs])

➤ Pathname compiled-file writes to if invoked with the same arguments.

(load path)

➤ Load source file or compiled file into Lisp environment. Return T if successful.

16 Virtual Machine

16.1 Load

load 

➤ Load source file or compiled file into Lisp environment. Return T if successful.

16.2 Compilation

compile

➤ Compile file.

16.3 Symbols

symbol-name symbol

➤ Name, package, property list, value, or function, respectively, of symbol. setfable.

(dictionary new-doc)

➤ Get/set documentation string of foo of given type.

nil

➤ False; the empty list; the empty type, subtype of every type; *standard-input*; *standard-output*; the global environment.

17 Libraries

17.1 Compiler

load

➤ Load source file or compiled file into Lisp environment.

common-lisp

➤ Exports the defined names of Common Lisp except for those in the keyword package.

common-lisp-user

➤ Current package after startup; uses package common-lisp.

keyword

➤ Contains symbols which are defined to be of type keyword.

18 General Functions

18.1 Paths

29

➤ Symbol name, and pathname matching name, respectively, of *default host*.

Physical file pathname corresponding to (possibly logical) pathname of directory type, List directory. path function or replace Logical list, path or name. pathname made by filling in components missing ( ].

name

➤ String that sufficiently describes the Canonical of name; or Time at which path

can be used.

default-pathname-defaults

➤ Pathname to use if one is needed and none supplied.

user-homedir-pathname [host] ❣ User’s home directory.

enough-namestring path-or-stream

➤ Return minimal path string that sufficiently describes the path of path-or-stream relative to root-path.

translate-pathname path-or-stream wildcard-path-a wildcard-path-b

➤ Translate the path of path-or-stream from wildcard-path-a into wildcard-path-b.

pathname path-or-stream

➤ Pathname of path-or-stream.

logical-pathname logical-path-or-stream

➤ Logical pathname of logical-path-or-stream. Logical pathnames are represented as all-uppercase wildcard to wildcard translations for logical-host. setfable.

load-logical-pathname-translations logical-host

➤ Load logical-host’s translations. Return NIL if already loaded; return T if successful.

translate-logical- pathname path-or-stream

➤ Physical pathname corresponding to (possibly logical) pathname of path-or-stream.

probe-file

➤ Canonical name of file. If file does not exist, return NIL/signal file-error, respectively.

true-name file

➤ Time at which file was last written.

file-write-date file

➤ Return name of file owner.

file-owner file

➤ Return length of stream.

file-length stream

➤ Rename file foo to bar. Unspecified components of path bar default to those of foo. Return new pathname, old physical file name, and new physical file name.

rename-file foo bar

➤ Delete file. Return T.

delete-file file

➤ List of pathnames matching path.
14 Packages and Symbols

The Loop Facility provides additional means of symbol handling; see loop, page 21.

14.1 Predicates

(symbolp foo)   ▷ T if foo is of indicated type.

(packagep foo)  ▷ T if foo is a package.

(keywordp foo)  ▷ T if foo is a keyword.

14.2 Packages

:bar | keyword:bar   ▷ Keyword, evaluates to :bar.

package:symbol ▷ Exported symbol of package.

package::symbol ▷ Possibly unexported symbol of package.

(defpackage foo
  
  (:nicknames nick*)
  (:documentation string)
  (:intern intern-symbol*)
  (:use used-package*)
  (:import-from pkg imported-symbol*)
  (:shadowing-import-from pkg shd-symbol*)
  (:shadow shd-symbol*)
  (:export exported-symbol*)
  (:size int))
▷ Create or modify package foo with interned-symbols, symbols from used-packages, imported-symbols, and shd-symbols. Add shd-symbols to foo’s shadowing list.

(make-package foo
  
  (:nicknames nick*)
  (:use (used-package*)))
▷ Create package foo.

(rename-package package new-name [new-nicknames])
▷ Rename package. Return renamed package.

(in-package foo)
▷ Make package foo current.

(use-package other-packages [package])
▷ Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T.

(package-use-list package)
▷ List of other packages used by/using package.

(delete-package package)
▷ Delete package. Return T if successful.

*packages
▷ The current package.

(list-all-packages) ▷ List of registered packages.

(package-name package) ▷ Name of package.

(package-nicknames package) ▷ Nicknames of package.

(find-package name) ▷ Package with name (case-sensitive).

(find-all-symbols foo) ▷ List of symbols foo from all registered packages.

14.3 Symbols

A symbol has the attributes name, home package, property list, and optionally value (of global constant or variable name) and function (function, macro, or special operator name).

(make-symbol name)
▷ Make fresh, uiintered symbol name.

(gensym) ▷ Return fresh, unintered symbol #\$n with n from *gensym-counter*. Increment *gensym-counter*.

(gentemp [prefix] [package]) ▷ Intern fresh symbol in package. Deprecated.

(copy-symbol symbol [props]) ▷ Return unintered copy of symbol. If props is T, give copy the same value, function and property list.
macroexpand-1) → Return macro expansion, once or entirely, respectively, of form and T if form was a macro form. Return form and NIL otherwise.

*macroexpand-hook* ➤ Function of arguments expansion function, macro form, and environment called by *macroexpand-1 to generate macro expansions.

trace {function [setf function]}
➤ Cause functions to be traced. With no arguments, return list of traced functions.

untrace {function [setf function]}
➤ Stop functions, or each currently traced function, from being traced.

trace-output
➤ Output stream ntrace and ntime send their output to.

step form
➤ Step through evaluation of form. Return values of form.

break [control arg]!
➤ Jump directly into debugger; return NIL. See page 36, *format, for control and args.

time
➤ Evaluate forms and print timing information to *trace-output*. Return values of form.

inspect foo
➤ Interactively give information about foo.

describe foo [stream standard-output]
➤ Send information about foo to stream.

describe-object foo [stream]
➤ Send information about foo to stream. Called by describe.

disassemble function
➤ Send disassembled representation of function to *standard-output*. Return NIL.

room [{NIL default | [stream]}
➤ Print information about internal storage management to *standard-output*.

15.4 Declarations

(proclaim decl)
➤ Globally make declaration(s) decl. decl can be: declaration, type, ftype, inline, notinline, optimize, or special. See below.

(declare decl*)
➤ Inside certain forms, locally make declarations decl*, decl can be: dynamic-extent, type, ftype, ignorable, ignore, inline, notinline, optimize, or special. See below.

(declaration foo*)
➤ Make foos names of declarations.

(dynamic-extent variable* (function function*)
➤ Declare lifetime of variables and/or functions to end when control leaves enclosing block.

(type type variable*)

(ftype type function*)
➤ Declare variables or functions to be of type.

(declare (ignore) {var (function function*)})
➤ Suppress warnings about used/unused bindings.

(inline function*)

(notinline function*)
➤ Tell compiler to integrate/not to integrate, respectively, called functions into the calling routine.

(compilation-speed)

(debug)
➤ Tell compiler how to optimize. n = 0 means unimportant, n = 1 is neutral, n = 3 means important.

(special var*)
➤ Declare vars to be dynamic.

16 External Environment

(get-internal-real-time)
➤ Current time, or computing time, respectively, in clock ticks.

(internal-time-units-per-second)
➤ Number of clock ticks per second.

(encode-universal-time sec min hour date month year [zone])

(get-universal-time)
➤ Seconds from 1900-01-01 00:00, ignoring leap seconds.

(decode-universal-time universal-time [time-zone])

(get-decoded-time)
➤ Return second, minute, hour, date, month, year, day, daylight-p, and zone.

(short-site-name)

(long-site-name)
➤ String representing physical location of computer.

(lisp-implementation)

(machine)
➤ Name or version of implementation, operating system, or hardware, respectively.

(machine-instance)
➤ Computer name.